Seek-Efficient I/O Optimization in Single Failure Recovery for XOR-Coded Storage Systems

Zhirong Shen, Jiwu Shu, *Member, IEEE*, Patrick P. C. Lee, and Yingxun Fu

Abstract—Erasure coding provides an effective means for storage systems to protect against disk failures with low redundancy. One important objective for erasure-coded storage systems is to speed up single disk failure recovery. Previous approaches reduce the amount of read data for recovery by reading only a small subset of data. However, they often incur high disk seeks, which may negate the resulting recovery performance. We propose *SIOR*, a seek-efficient I/O recovery algorithm for improving the performance of single disk failure recovery. SIOR carefully balances the trade-off between the amount of read data and the number of disk seeks by considering the data layout at the multi-stripe level. It then greedily determines the data to read for recovery using Tabu search. Experiments show that SIOR achieves similar performance to the brute-force enumeration method while keeping high search efficiency. Also, SIOR reduces $31.8 \sim 65.1$ percent of disk seeks during recovery and provides up to 150.0 percent recovery speed improvement, when compared to a state-of-the-art greedy recovery approach.

Index Terms—XOR-Coded storage systems, single failure fecovery, disk seeks, repair traffic, greedy algorithm

1 Introduction

Disk failures are commonplace in large-scale storage systems [2], and hence protecting data with redundancy is critical to provide fault tolerance guarantees. Given that replication has high storage overhead and aggravates the operational cost of storage, *erasure coding* is increasingly adopted in commercial storage systems (e.g., Azure [3] and Facebook [4], [5]). The core idea of erasure coding is to take original pieces as inputs and encode them into redundant pieces, where the original and redundant pieces collectively form a *stripe*, such that the original pieces can be reconstructed from any sufficient number of original/redundant pieces within a stripe (see Section 2.1 for details). A storage system organizes data in different stripes, each of which is independently encoded and operated by erasure coding. It distributes the original/redundant pieces of each stripe across different disks, so as to tolerate multiple disk failures.

Although erasure coding tolerates multiple disk failures, single disk failures dominate over 90 percent of failure events in practice [2], [4], [6]. Conventional recovery for single (disk) failures often needs to read a considerable amount of surviving data. To reduce the amount of read data during single failure recovery, researchers propose a spate of solutions [6], [7], [8], [9], [10], [11], which can be categorized as follows:

1) New constructions of RAID-6 codes that tolerate double failures, such as HDP Code [10], F-MSR Code [12], HV Code [11], which read less data for recovery than existing RAID-6 codes.
2) Optimization techniques for specific codes, such as RDP Code (by [7]) and X-Code (by [13]), which provide a provably lower bound of the amount of read data.
3) General optimization techniques for any erasure codes [6], [9], [14].

Although these studies effectively reduce the amount of read data for recovery, they achieve this by reading and correlating different portions of a stripe to reconstruct the failed data. This leads to additional *disk seeks*,[1] which can negate the actual recovery performance. A disk seek needs to move the mechanic disk head to the position where the accessed data resides, and its overhead is generally larger than that of transferring the same size of accessed data. However, most existing studies do not take into account the disk seek overhead in their recovery solutions. Some studies mitigate the disk seek overhead by assuming large-size I/O units (e.g., 16 MB [6], [9], [11], [14], [16], [17]), yet traditional file systems often operate on small-size disk blocks (e.g., $4 \sim 16$ KB [18], [19], [20]). How to balance the trade-off of different block sizes between normal I/O performance and recovery performance remains unexplored.

In this paper, we examine the seek-efficient I/O optimization problem of single failure recovery for any XOR-based erasure code, whose encoding and decoding operations are purely XOR operations. Our observation is that previous studies only focus on the recovery at the *single* stripe level, yet storage systems typically organize data in stripes [21]. Thus, by collectively examining the data layouts of multiple stripes, we can reduce the number of disk seeks in recovery. To this end, we propose *SIOR*, a Seek-efficient I/O Recovery algorithm for a multi-stripe single failure recovery, such that it not only reduces the amount of read data, but also reduces the number

• *Z. Shen, J. Shu, and Y. Fu are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: zhirong.shen2601@gmail.com, shujw@tsinghua.edu.cn, mooncape1986@126.com.*
• *P.P.C. Lee is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong. E-mail: pclee@cse.cuhk.edu.hk.*

Manuscript received 23 Jan. 2016; revised 28 May 2016; accepted 1 July 2016. Date of publication 13 July 2016; date of current version 15 Feb. 2017. Recommended for acceptance by M. Steinder.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2591040

1. We use the term "seeks" to collectively refer to seeks and rotations [15].

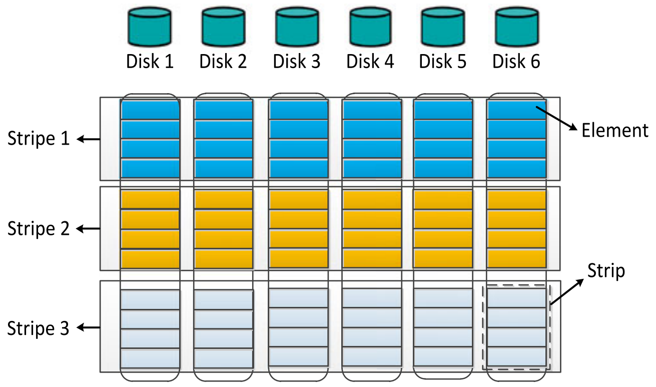


Fig. 1. Relationships among disk, stripe, strip, and element in an XOR-coded storage system.

of disk seeks. SIOR is suitable for the storage systems that are sensitive to both repair traffic (i.e., the amount of read data for recovery) and disk seeks for the data reconstruction.

SIOR can be decomposed into two stages. In the first stage, SIOR finds an initial recovery solution that can rebuild the lost data at the multi-stripe level and satisfies the requirements on the data reduction. In the second stage, SIOR utilizes Tabu Search [22] to iteratively obtain another solution that achieves fewer disk seeks without violating the requirement on the data reduction. In addition, SIOR uses a *filling method* that selectively reads unrequested data to further reduce disk seeks. Therefore, SIOR fully lightens the load on disks during single failure recovery, in terms of the amount of read data and the number of disk seeks.

In summary, we make the following contributions.

1) We formulate the seek-efficient I/O optimization problem for single failure recovery. The formulation aims to reduce both the amount of read data and the number of disk seeks.
2) We leverage Tabu search [22] and propose a general greedy algorithm called SIOR that is applicable for any XOR-based erasure code. SIOR iteratively selects a recovery solution with fewer disk seeks, while preserving the reduction on the amount of read data for recovery. SIOR also uses a filling method to reduce disk seeks.
3) We implement SIOR over a storage system testbed constructed from several representative XOR-based erasure codes. Evaluation shows that SIOR significantly reduces the search time when compared to the brute-force enumeration method. In addition, we compare SIOR with Zpacr [14], a state-of-the-art greedy recovery approach that minimizes the amount of read data but does not consider disk seeks. We show that SIOR reduces $31.8 \sim 65.1$ percent disk seeks during the recovery, and provides up to 150.0 percent recovery speed improvement.

The rest of this paper is organized as follows. Section 2 introduces the research background. The problem formulation is provided in Section 3. We present the detailed design of SIOR in Section 4. After that, we evaluate the performance of SIOR in Section 5 and conclude our work in Section 6.

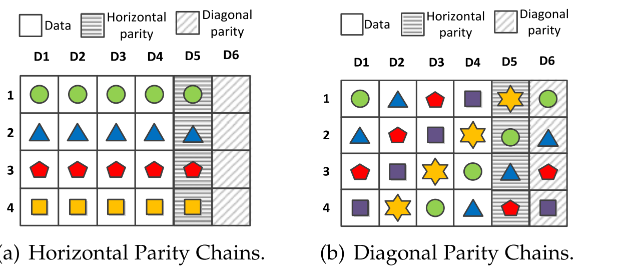


Fig. 2. A stripe of RDP Code with $p+1$ disks ($p=5$). $\{E_{1,1}, E_{1,2}, \ldots, E_{1,5}\}$ is a horizontal parity chain and $\{E_{1,1}, E_{4,3}, E_{3,4}, E_{2,5}, E_{1,6}\}$ is a diagonal parity chain.

2 Background

2.1 Basics

One typical class of erasure codes is **M**aximum **D**istance **S**eparable (MDS) codes, which reach the optimal storage efficiency for a given level of fault tolerance. MDS codes are typically configured by two parameters k and m: an (k, m) MDS code transforms k pieces of original data pieces and encodes them to produce another m redundant pieces, such that *any* k out of $k+m$ pieces are sufficient to reconstruct the original data. The $k+m$ dependent pieces collectively form a *stripe* and are distributed to $k+m$ disks.

A popular family of MDS codes is XOR-based erasure codes [8], [10], [11], [14], [23], [24], [25], [26], whose encoding/decoding operations are purely based on XOR operations, so as to achieve fast parity calculation and data reconstruction. This type of erasure codes generally achieve better encoding, decoding, and update performance than the erasure codes that are based on complicated operations in the finite field, such as Reed-Solomon Code [27], LRC Code [3], and STAIR Code [28]. XOR-based erasure codes have been widely used in current storage systems [29], [30], [31]. In this paper, we call the storage systems that are protected by XOR-based erasure codes to be "XOR-coded storage systems".

An erasure-coded storage system is composed of many independent stripes. Fig. 1 shows the logical structure of an XOR-coded storage system with three stripes. Meanwhile, we denote the maximum set of information in a stripe that stored on the same disk as *strip*. In XOR-coded storage systems, a strip is partitioned into w equal-size elements, where an *element* is the basic operation unit (e.g., byte or block) in XOR-coded storage systems. For example, in Fig. 1, each stripe has six strips and every strip has four elements.

To clarify the relationships among the elements in a stripe, Fig. 2 presents the layouts of RDP Code [25] in a stripe. RDP Code is a typical RAID-6 code for double fault tolerance and is constructed over $p+1$ disks, where p is a prime number used

TABLE 1
Background Notations and Descriptions

Notations	Descriptions
p	prime number to configure the stripe size
n	number of disks in an XOR-coded storage system
k, m	number of data disks, number of parity disks
w	number of elements in a strip
D_j	the jth disk
$E_{i,j}$	element at the ith row and jth disk
$\oplus$	XOR operation
$\sum\{E_{i,j}\}$	sum of XOR operation among the elements $\{E_{i,j}\}$

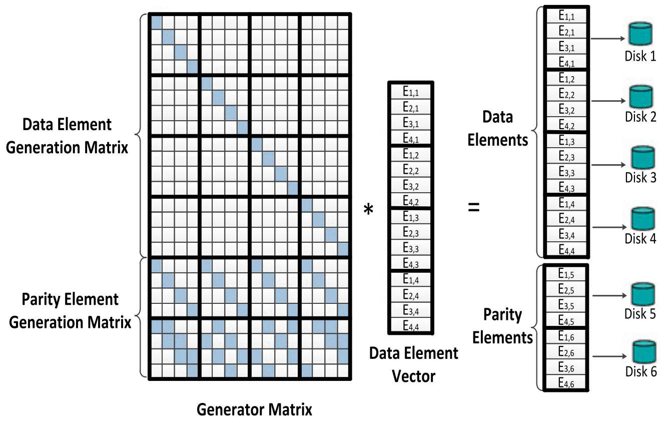


Fig. 3. The generator matrix for RDP Code ($p = 5$) in a stripe. The filled cell in the generator matrix means "1", while the blank cell means "0".

for configuring the number of disks in a stripe. Table 1 also lists the notations and their descriptions. Specifically, a stripe usually consists of *data elements* and *parity elements*. Data elements contain the original data information, while parity elements keep the redundant information. For example, $E_{1,1}$ is a data element and $E_{1,5}$ is a parity element in Fig. 2a.

To generate a parity element, a subgroup of data elements will be selected to perform the encoding operation by using XOR operations. For example, RDP Code needs to calculate "horizontal parity elements" and "diagonal parity elements". The horizontal parity element $E_{1,5} := \sum_{i=1}^{4} E_{1,i} := E_{1,1} \oplus \cdots \oplus E_{1,4}$ in Fig. 2a, and the diagonal parity element $E_{1,6} := E_{1,1} \oplus E_{4,3} \oplus E_{3,4} \oplus E_{2,5}$ as shown in Fig. 2b.

We also denote the parity generation relationship as the *parity chain* when it is used to repair lost elements. A parity chain consists of a parity element and the associated elements in that parity element's generation. For example, as shown in Fig. 2, a horizontal parity element $E_{1,5}$ is generated based on elements $\{E_{1,1}, E_{1,2}, \ldots, E_{1,4}\}$. Then the set of elements $\{E_{1,1}, E_{1,2}, \ldots, E_{1,5}\}$ constitutes a parity chain. Once an element fails, it can be recovered by using the surviving elements in the associated parity chains. For example, $E_{1,1}$ can be recovered by two parity chains in Fig. 2, i.e., $\{E_{1,1}, E_{1,2}, \ldots, E_{1,5}\}$ that generates the horizontal parity element $E_{1,5}$ and $\{E_{1,1}, E_{4,3}, E_{3,4}, E_{2,5}, E_{1,6}\}$ that produces the diagonal parity element $E_{1,6}$.

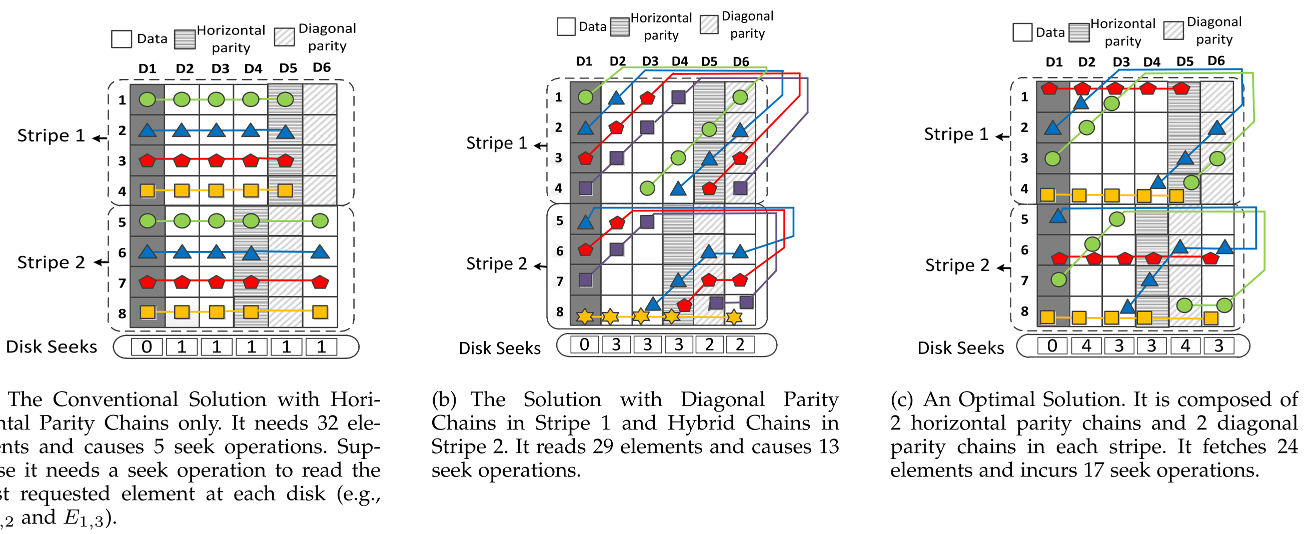
To represent any XOR-based erasure code, one can arrange the data elements residing on disks into a *data element vector* and utilize a *generator matrix* [32], which is actually a binary matrix, to realize the encoding procedure as the multiplication between the generator matrix and the data element vector. For example, Fig. 3 illustrates the generator matrix of RDP Code ($p = 5$) in a stripe.

2.2 Single Failure Problem

For single failure recovery, most previous works propose to accelerate the repair process by reducing the number of elements that need to be read from the surviving disks.

For RDP Code, Xiang et al. [7] first propose to repair the corrupted elements by selectively using the diagonal parity chains and the horizontal parity chains, so that the optimal solution with the fewest retrieved elements will be obtained. Fig. 4 shows three recovery schemes to rebuild Disk 1 in two stripes for RDP Code. To repair the lost data elements in the Stripe 1, the first two schemes (i.e., Figs. 4a and 4b) only use horizontal parity chains and diagonal parity chains respectively during the recovery, and both of the schemes require 16 elements. On the other hand, if we mix the two kinds of parity chains, the recovery solution of Stripe 1 as shown in Fig. 4c, we only need 12 elements. Following this inspiration, Xu et al. [13] investigate the optimal recovery for single failure in X-Code [8]. Many erasure codes [3], [10], [11] also consider the performance of single failure recovery specifically in their code constructions.

To minimize the number of read elements during single failure recovery for *any* XOR-based erasure code, Khan et al. [6] propose to enumerate all the parity chains for each failed element based on the generator matrix and convert the possible parity chains into a directed and weighted graph. Therefore, the optimization of single failure recovery can be transformed to the problem of finding the shortest path in the converted graph. Besides retrieving the fewest elements, Luo et al. [9] also show that the solution that evenly reads elements from surviving disks can achieve a faster recovery. Fu et al. [16] further study the load-balancing problem at



(a) The Conventional Solution with Horizontal Parity Chains only. It needs 32 elements and causes 5 seek operations. Suppose it needs a seek operation to read the first requested element at each disk (e.g., $E_{1,2}$ and $E_{1,3}$).

(b) The Solution with Diagonal Parity Chains in Stripe 1 and Hybrid Chains in Stripe 2. It reads 29 elements and causes 13 seek operations.

(c) An Optimal Solution. It is composed of 2 horizontal parity chains and 2 diagonal parity chains in each stripe. It fetches 24 elements and incurs 17 seek operations.

Fig. 4. The three recovery solutions to repair Disk 1 for RDP Code ($p = 5$) in two rotated stripes. It indicates that the data reduction in single failure recovery may increase the seek operations.

multi-stripe setting. However, all of the above solutions involve NP-Hard problems and need to spend a significantly long search time to obtain the optimal solution. Thus, they cannot be easily generalized to support on-line reconstruction, which determines the optimal recovery solution on-the-fly based on the current system configurations. Aiming at this shortcoming, Zhu et al. [14] greedily search the solution with a near-minimum number of read elements for any XOR-based erasure code.

2.3 Open Problems

Although there have been extensive efforts on single failure recovery, there remain two open problems.

The Increasing Number of Disk Seeks. Most prior works [6], [7], [9], [10], [11], [14], [16], [33] only focus on reducing the number of elements to be read during the recovery, but neglect the increasing number of disk seeks. For example, in Fig. 4a, it reads 32 elements from the surviving disks, and triggers five seek operations. Fig. 4c only reads 24 elements, but produces 17 seek operations. Disk seeks are expensive operations, especially when the size of an I/O unit is small. Thus, the optimization on disk seeks should be carefully studied for single failure recovery.

Recovery in Multiple Stripes. Most previous studies [6], [7], [9], [10], [11], [33] only consider the recovery in a single stripe, mainly because their problems are stripe-independent and they can directly generalize the stripe-level solutions for multiple stripes. However, the optimization of disk seeks is more complicated, as the number of disk seeks usually correlates with the positions of read elements among stripes. For example, in Fig. 4b, the two elements $E_{4,3}$ (in Stripe 1) and $E_{5,3}$ (in Stripe 2) can be read directly, while it needs an extra seek operation to retrieve $E_{3,2}$ (in Stripe 1) and $E_{5,2}$ (in Stripe 2). Therefore, it is more practical to consider this problem at the multi-stripe setting.

In practice, as the disk hosting parity elements will usually serve the heavy updates when the associated data elements are written, multiple stripes are usually organized by the stripe rotation [21] for load balancing. In this paper, we rotate the stripes by gradually shifting the elements to left when the stripe identity increases. Take Fig. 4 as an example. Parity elements of Stripe 1 are placed at D_5 and D_6, and parity elements of Stripe 2 will be shifted to D_4 and D_5.

3 Problem Formulation

3.1 Model

This paper addresses the following fundamental question: Can we mitigate the disk seek overhead during single failure recovery, while still reducing the amount of read data? To address the question, we formulate the following problem: Given an expected number of retrieved elements, our goal is to minimize the number of disk seeks at the multi-stripe level.

We now formulate the problem as follows. Suppose an XOR-coded storage system consists of n disks $\{D_1, \ldots, D_n\}$. Given a corrupted disk D_i $(1 \le i \le n)$ and the expected number of retrieved elements M, suppose that the recovery solution reads m_j elements from disk D_j $(1 \le j \ne i \le n)$ and consequently causes o_j seek operations. Our objective is to find the recovery solution that minimizes the number of seek operations and also retrieves no more than M elements. This objective can be formulated as follows:

$$\text{Minimize} \sum_{1 \le j \ne i \le n} o_j, \tag{1}$$

subject to

$$\sum_{1 \le j \ne i \le n} m_j \le M. \tag{2}$$

3.2 Assumptions

We assume that a storage system treats an element as a storage block on disk, such that the blocks serve as not only the encoding/decoding units of a given XOR-based erasure code, but also the read/write units of a storage system. This treatment has been extensively assumed by previous studies on I/O-efficient recovery of XOR-based erasure codes (e.g., [6], [7], [10], [11], [13], [14], [34]). Previous studies often assume a large block size to mitigate the overhead due to disk seeks, while our work relaxes this assumption by taking into account the disk seek overhead in our recovery solution.

Our work is applicable for both on-line and off-line reconstructions [35], in which the former continually serves users' I/O requests when recovering lost data, while the latter uses all available resources to recover lost data only without handling foreground I/O requests. Our focus is to mitigate the impact of disk seeks during failure recovery, so we expect that both on-line and off-line reconstructions can benefit from our designs. On the other hand, the disk seeks in on-line reconstruction scenario are caused by both I/O activities (in the foreground) and the reconstruction process (in the background). How to leverage foreground I/O activities to further improve the performance of on-line reconstruction will be posed as future work.

3.3 Motivation

Our objective is an optimization problem of minimizing the number of disk seeks with a constraint on the number of read elements. To achieve this objective, the *enumeration method* that exhaustively tests all possible parity chains for each lost element will cause an extremely high computation complexity. For example, consider the case where a disk fails in X-Code [8]. Suppose that there are s stripes and the prime number is p. Then there are $(p-2)s$ data elements on each disk and each lost data element can be recovered by at least two parity chains (according to the property of RAID-6), so the search space is larger than $2^{(p-2)s}$. Pre-storing the optimal recovery solution cannot always retain its optimality, especially under the environment with constant changing factors, for example, the changing available network bandwidth in a heterogeneous environment [33].

Therefore, a better idea is to propose a greedy algorithm that can efficiently pursue a near-optimal solution. To this end, we propose to partition the search procedure into two stages. In the first stage, we find an "initial solution" that satisfies the restriction (2). As discussed above, extensive studies have been made to minimize the number of required elements in a single stripe, such as Zpacr [14]. Therefore, we can apply any existing approach for each stripe in turn, so as to obtain an initial solution. In the

TABLE 2
The Used Symbols and Descriptions in SIOR

Symbols	Descriptions
	Defined in **Algorithm 1**
M	expected number of read elements for recovery
$\mathbb{R}_{ini}$	initial recovery solution of s stripes
$\mathcal{R}_i$	recovery solution for the ith stripe before optimization
$\overline{\mathcal{R}}_i$	recovery solution for the ith stripe after optimization
s	number of stripes
$\mathbb{D}$	distribution of read elements before filling
$e(\cdot)$	function to calculate the number of read elements
	Defined in **Algorithm 2**
$\mathbb{D}'$	candidate distribution of read elements before filling
$\mathbb{F}'$	candidate distribution of read elements after filling
I	an interval to be filled
$\lvert I\rvert$	number of elements included in I
$\mathcal{S}_I$	set of disks that have the intervals with size $\lvert I\rvert$
D_I	disk that has the lightest load in $\mathcal{S}_I$
	Defined in **Algorithm 3**
$\mathcal{L}$	Tabu list
$\mathbb{F}_{opt}$	optimal recovery solution found by SIOR
C_x	candidate parity chain for the lost element x
S_x	selected parity chain for the lost element x
$\mathcal{A}$	set of candidate solutions in an iteration
$q(\cdot)$	function to calculate the number of disk seeks

second stage, we can further perform more fine-grained optimization based on the initial solution, by trying every other possible parity chain for each lost element and greedily selecting the best replacement in each iteration.

4 The Detailed Design of SIOR

Based on the above motivation, we propose a greedy algorithm called SIOR. SIOR can be decomposed into "InitialSolutionSelection" and "InitialSolutionOptimization". Table 2 lists the used notation and descriptions in SIOR.

4.1 Initial Solution Selection

To obtain an initial solution, for each stripe, we can make use of existing stripe-level[2] methodologies [6], [14] to minimize the number of read elements for recovery, until the number of retrieved elements accumulated in all the stripes satisfies the constraint (2). There exist two methodologies that address single failure recovery for any XOR-based erasure code, including the enumeration scheme [6], which is shown to be NP-Hard, and the greedy algorithm Zpacr [14], which finds the near-optimal solution in polynomial time. To efficiently provide an on-line recovery solution, we choose Zpacr [14] to serve as the cornerstone in InitialSolutionSelection. Algorithm 1 presents the detailed procedure.

Algorithm Details. At the beginning, we first initialize a solution $\mathcal{R}_i$ $(1 \leq i \leq s)$ for the ith stripe (steps $1 \sim 2$), where $\mathcal{R}_i$ is composed of parity elements. It indicates that the lost elements in the ith stripe can be recovered by using the parity chains that associate with these parity elements in $\mathcal{R}_i$. For example, $\mathcal{R}_1 := \{E_{1,5}, \ldots, E_{4,5}\}$ in Stripe 1 of Fig. 4a, where the lost element $E_{j,1}$ $(1 \leq j \leq 4)$ can be recovered by using the horizontal parity chain that generates $E_{j,5}$, i.e., $E_{j,1} = E_{j,2} \oplus E_{j,3} \oplus E_{j,4} \oplus E_{j,5}$.

2. The stripe-level means that these methodologies can only optimize the number of retrieved elements for a single stripe.

Algorithm 1. Initial Solution Selection

```
Input: $s$: number of stripes
       $M$: expected number of read elements in $s$ stripes
Output: $\mathbb{R}_{ini}$: initial solution
1  for $i = 1$ to $s$ do
2      Initialize $\mathcal{R}_i$ for the $i$th stripe
3  Build a recovery solution $\mathbb{R}_{ini} := (\mathcal{R}_1, \ldots, \mathcal{R}_s)$
4  for $i = 1$ to $s$ do
5      Apply Zpacr [14] to $\mathcal{R}_i$ and get $\overline{\mathcal{R}}_i$
6      Set $\mathbb{R}_{ini} := (\overline{\mathcal{R}}_1, \ldots, \overline{\mathcal{R}}_i, \mathcal{R}_{i+1}, \ldots, \mathcal{R}_s)$
7      Get the element distribution $\mathbb{D}_{ini}$ of $\mathbb{R}_{ini}$
8      if $e(\mathbb{D}_{ini}) \leq M$ then
9          Return $\mathbb{R}_{ini}$
10 Return false
```

In this paper, a recovery solution is said to be *valid* if it satisfies the constraint (2) on the number of retrieved elements. We construct an initial solution $\mathbb{R}_{ini}$ that includes the recovery solutions for s stripes (step 3). Then for the ith stripe, we take $\mathcal{R}_i$ as input, apply Zpacr [14] for optimization, and obtain a near-optimal recovery solution $\overline{\mathcal{R}}_i$ (step 5). We replace $\mathcal{R}_i$ with $\overline{\mathcal{R}}_i$ in $\mathbb{R}_{ini}$ and get an updated initial solution $\mathbb{R}_{ini} := (\overline{\mathcal{R}}_1, \ldots, \overline{\mathcal{R}}_i, \mathcal{R}_{i+1}, \ldots, \mathcal{R}_s)$ (step 6). Suppose that $\mathbb{D}_{ini}$ denotes the element distribution of $\mathbb{R}_{ini}$ and $e(\mathbb{D}_{ini})$ represents the number of retrieved elements in $\mathbb{D}_{ini}$ (step 7). Once the total number of retrieved elements $e(\mathbb{D}_{ini})$ is no more than the expected number of read elements M, $\mathbb{R}_{ini}$ will be returned as a valid initial solution (steps $8 \sim 9$). Otherwise, the algorithm will return false (step 10).

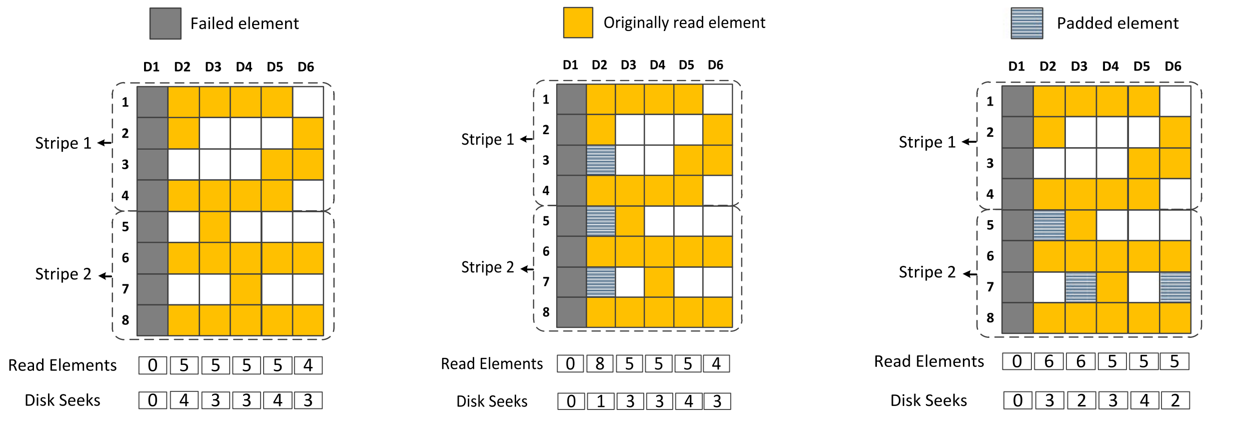
An Example. We take Fig. 4 as an example, where $s = 2$, and M is set as 24. We use a parity element to represent the associated parity chain for data reconstruction. Algorithm 1 first generates a solution $\mathbb{R}_{ini} := (\mathcal{R}_1, \mathcal{R}_2)$ and its element distribution $\mathbb{D}_{ini}$ is shown in Fig. 4a, where $\mathcal{R}_1 := \{E_{1,5}, E_{2,5}, \ldots, E_{4,5}\}$ and $\mathcal{R}_2 := \{E_{5,4}, E_{6,4}, \ldots, E_{8,4}\}$. It reads 32 elements. After performing the optimization, we obtain the initial solution $\mathbb{R}_{ini} := (\overline{\mathcal{R}}_1, \overline{\mathcal{R}}_2)$ and its distribution $\mathbb{D}_{ini}$ is shown in Fig. 4c, where $\overline{\mathcal{R}}_1 := \{E_{1,5}, E_{2,6}, E_{3,6}, E_{4,5}\}$ and $\overline{\mathcal{R}}_2 := \{E_{6,5}, E_{6,4}, E_{8,5}, E_{8,4}\}$. The initial solution retrieves 24 elements and thus satisfies the constraint (2).

4.2 Initial Solution Optimization

Although the initial solution satisfies the constraint (2), it may trigger many disk seeks. We now propose an algorithm based on Tabu Search [22] to optimize the number of disk seeks. Tabu Search effectively prevents a search from being confined in suboptimal search regions when compared to other local search methods like hill-climbing algorithm [14]. It specifies a search rule and keeps a structure called a Tabu list. For a visited solution, once it violates the rule, it will be recorded in the Tabu list and passed over for a period of time. Before explicitly presenting our detailed algorithm, we first give a rough sketch about its main idea.

Main Idea:

1) *Simplified recovery model*. To iteratively make the initial solution approach the optimal one, an important step



(a) Element Distribution Before Filling. It needs 24 elements and suppose $M = 27$. The recovery solution causes 17 disk seeks.

(b) An Example of Element Distribution After Filling. The recovery solution causes 14 disk seeks. Disk 2 is the most loaded disk and it serves 8 element retrievals.

(c) An Example of Element Distribution After Filling. The recovery solution needs 14 disk seeks. Disk 2 is the most loaded disk and it serves 6 element retrievals.

Fig. 5. Making filling procedure more balanced.

is to try other parity chains to repair lost elements and then execute a greedy selection. For example, in the solution of Fig. 4a, the lost data element $E_{1,1}$ is recovered by the horizontal parity chain that generates $E_{1,5}$. We can obtain another recovery solution by simply selecting another parity chain that $E_{1,1}$ involves, i.e., the diagonal parity chain that produces $E_{1,6}$. Some previous works, such as [6] and [9], adopt the enumeration method to exhaustively try every possible parity chain according to the generator matrix. Unfortunately, this enumeration problem is NP-Hard.

Given the need for efficiency, we choose a simplified recovery model instead. We repair lost elements by simply using parity chains that generate the parity elements. For example, we only consider the horizontal/diagonal parity chains for data recovery in RDP Code, as it only has these two kinds of parity elements. Following the same principle, we only use the diagonal/anti-diagonal parity chains for data reconstruction in X-Code.

2) *The filling procedure*. Algorithm 1 may return a recovery solution that requests less than M elements. Given the distribution of requested elements, we first define the concept of an *interval*, which enables us to measure disk seeks.

Definition 1. *If two sequentially requested elements are not physically contiguous, we call the maximum set of unrequested elements between them to be an "interval".*

We take the solution illustrated in Fig. 5a as an example. $E_{4,2}$ and $E_{6,2}$ are two elements that are sequentially requested, and $E_{5,2}$ is an element that is not needed between them. Thus, $E_{5,2}$ is called an interval, since it takes an extra seek operation for the disk to read $E_{4,2}$ and $E_{6,2}$. Similarly, the set $\{E_{2,3}, E_{3,3}\}$ forms an interval between the requested elements $E_{1,3}$ and $E_{4,3}$. The idea of the *filling procedure* is to read all the unrequested elements in the selected intervals, so as to reduce the number of disk seeks. For example, the solution in Fig. 5a needs 17 disk seeks. As a comparison, the solution in Fig. 5c reads the same requested elements in Fig. 5a, but additionally reads the unrequested elements (i.e., $E_{5,2}$, $E_{7,3}$, and $E_{7,6}$). Thus, it reduces the number of disk seeks reduces to 14. We can observe that we can "fill" an interval by reading the elements in it, so as to save one seek operation.

To generalize the idea, suppose that we are given a valid recovery solution $\mathbb{R}$ and its element distribution $\mathbb{D}$, and that the number of retrieved elements in $\mathbb{D}$ is $e(\mathbb{D})$. To reduce the number of disk seeks, we can fill the intervals by retrieving no more than $M - e(\mathbb{D})$ elements that are unrequested. Theorem 1 shows how to reduce the maximum number of disk seeks for a given constant number of filled elements.

Theorem 1. *The filling procedure that fills the intervals in the order of smallest to largest in size (in terms of the number of elements included in an interval) can reduce the maximum number of seek operations.*

Proof. Since filling an interval can save one seek operation, filling more intervals will reduce more disk seeks. Given a constant number of available elements for filling, the method that fills the intervals in the order of smallest to largest in size can reduce the maximum number of intervals, and thus bring the most reduction of seek operations. □

Although the filling procedure can reduce the number of seek operations, it will change the distribution of elements to be retrieved, leading to unbalanced element retrievals across the surviving disks. The unbalanced load is undesirable, since it not only extends the recovery time [9], [16], but also easily makes the most loaded disk exhausted [10], [11]. Fig. 5 gives an example to show how the filling procedure causes an uneven distribution of element retrievals for recovery if it is not carefully designed. Fig. 5a first shows the distribution of elements to be read in Fig. 4c. It reads 24 elements. Suppose that $M = 27$. Fig. 5b only fills the intervals at Disk 2, while Fig. 5c respectively fills three intervals at Disk 2, Disk 3, and Disk 6. We can see that the distribution in Fig. 5c is more balanced than that in Fig. 5b.

Based on above observations, in addition to reducing the maximum number of intervals, we also propose to perform the filling procedure according to the element distribution and try to obtain a more balanced recovery solution. Our *main idea* is that given an interval to be filled, we always give a higher priority to the disk with a lighter load. Algorithm 2 describes how the filling procedure is performed when taking into account load balancing.

Algorithm 2. Filling Procedure

```
Input: $M$: number of read elements to repair $s$ stripes.
       $\mathbb{D}'$: candidate distribution before filling procedure.
Output: $\mathbb{F}'$: candidate distribution after filling procedure.
 1 Calculate $e(\mathbb{D}')$ and set $\Delta \leftarrow M - e(\mathbb{D}')$
 2 Sort the intervals from smallest to largest in size
 3 for each interval $I$ to be filled do
 4   $\Delta \leftarrow \Delta - |I|$
 5   if $\Delta < 0$ then
 6     Break
 7   Scan the surviving disks and obtain the disk set $\mathcal{S}_I$
 8   Select $D_I \leftarrow \mathcal{S}_I$
 9   Fill the interval at $D_I$
10 Obtain $\mathbb{F}'$ after filling procedure
11 Return $\mathbb{F}'$
```

Algorithm Details. Given a distribution of the elements to be read $\mathbb{D}'$, we first calculate $e(\mathbb{D}')$, which denotes the number of elements to be retrieved in $\mathbb{D}'$, and then obtain Δ (step 1). Δ denotes the maximum number of unrequested elements that can be read, so that the final recovery solution will not violate the constraint (2). To fill the most number of intervals, we first sort the intervals in $\mathbb{D}'$ in ascending order, and scan the interval from smallest to largest in size (step 2). For each interval I, suppose that its size is $|I|$. We first judge if the number of unrequested elements that is allowed to be read is enough to fill I. We terminate the filling procedure if the seek operations cannot be further reduced (steps $4 \sim 6$). To fill the interval I, we then scan every surviving disk and obtain a set of disks $\mathcal{S}_I$ that have intervals with size $|I|$ (step 7). We select a disk D_I that has the lightest load from $\mathcal{S}_I$ and perform the filling procedure at D_I (steps $8 \sim 9$). Finally, we can obtain a new element distribution $\mathbb{F}'$ once Δ cannot fill any interval (step 10).

An Example. Fig. 5 shows an example of how the filling procedure is performed. Based on the recovery solution in Figs. 4c and 5a first illustrates the distribution of the requested elements whose number is 24 (i.e., $e(\mathbb{D}') = 24$). Suppose $M = 27$, and the number of elements to be filled is $\Delta = M - 24 = 3$. To perform the filling procedure, Algorithm 2 first sorts the intervals from smallest to largest in size. According to Theorem 1, one has to fill the intervals in ascending order, such that the maximum number of disk seeks can be reduced. To pad the interval (i.e., I) whose size is 1, Algorithm 2 then selects the disk set $\mathcal{S}_I = \{D_2, D_3, D_4, D_5, D_6\}$, in which all the disks have the interval with size one. To make the distribution more balanced, we select D_6 (i.e., D_I), which has the minimum number of requested elements and fill the interval (i.e., reading the unrequested element $E_{7,6}$). After this filling procedure, Δ will be updated to two. Following this filling principle, we also fill another two intervals by reading $E_{5,2}$ and $E_{7,3}$ in the next filling procedure. Finally, we can obtain a new distribution of requested elements (i.e., $\mathbb{F}'$), as shown in Fig. 5c, after filling three intervals.

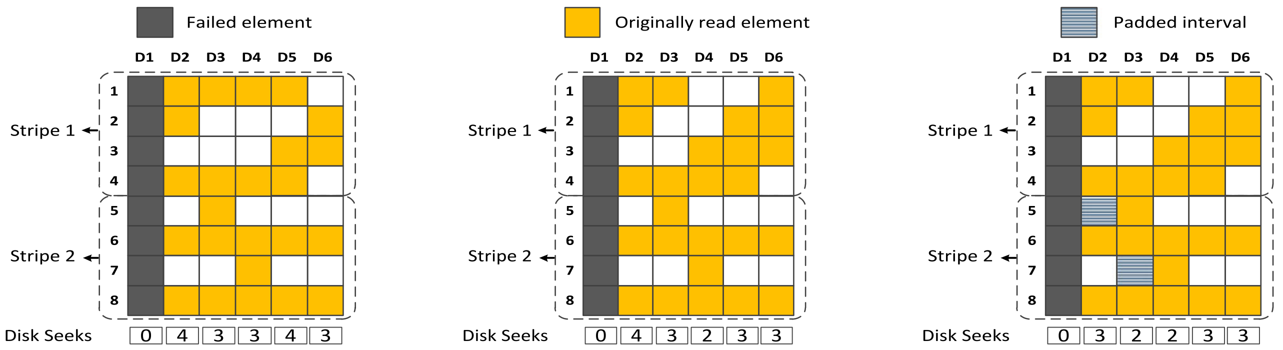
3) *The maintenance of a Tabu list*. We now elaborate how we optimize our initial solution, with an objective of reducing the number of disk seeks, using Tabu Search. Algorithm 3 presents the pseudo-code. It maintains a Tabu list $\mathcal{L}$ throughout the search, which keeps track of the number of seek operations of the selected solutions in recent iterations. In each iteration, Algorithm 3 will ignore the solutions that need the same disk seeks recorded in $\mathcal{L}$, and choose the one with the least disk seeks among the remaining candidates. This design effectively prevents the search process from being stuck to a suboptimal solution.

Algorithm 3. Initial Solution Optimization

```
Input: $\mathcal{L}$: Tabu list
       $t$: number of iterations
       $\mathbb{R}_{ini}$: initial recovery solution
       $M$: expected number of read elements to repair $s$
       stripes
Output: $\mathbb{F}_{opt}$: recovery solution found by SIOR
 1 Set $\mathcal{L} \leftarrow \emptyset$, $\mathbb{R} \leftarrow \mathbb{R}_{ini}$, $q(\mathbb{F}_{opt}) \leftarrow \infty$, $\mathcal{A} \leftarrow \emptyset$
 2 for each iteration do
 3   for each lost element $x$ do
 4     for each candidate parity chain $C_x$ do
 5       $\mathbb{R}' \leftarrow \mathbb{R} - S_x + C_x$
 6       if $\mathbb{R}'$ is not valid then
 7         Reuse Initial Solution Selection to optimize it
 8       Get the element distribution $\mathbb{D}'$
 9       Run Filling Procedure to $\mathbb{D}'$ and obtain $\mathbb{F}'$
10       Calculate the caused disk seeks $q(\mathbb{F}')$
11       if $q(\mathbb{F}') \notin \mathcal{L}$ then
12         Record $\{\mathbb{R}', \mathbb{F}'\}$ in $\mathcal{A}$
13   Select $\{\mathbb{R}_{min}, \mathbb{F}_{min}\}$ from $\mathcal{A}$
14   ▷ record the near − optimal scheme
15   if $q(\mathbb{F}_{min}) \leq q(\mathbb{F}_{opt})$ then
16     $q(\mathbb{F}_{opt}) \leftarrow q(\mathbb{F}_{min})$
17     $\mathbb{F}_{opt} \leftarrow \mathbb{F}_{min}$
18   ▷ update the Tabu list
19   if $\mathcal{L}$ is full then
20     Evict the oldest value out of $\mathcal{L}$
21   Append $q(\mathbb{F}_{min})$ to $\mathcal{L}$
22   Update $\mathbb{R} \leftarrow \mathbb{R}_{min}$, set $\mathcal{A} \leftarrow \emptyset$
23 Return $\mathbb{F}_{opt}$
```

Algorithm Details. In Algorithm 3, we first initialize a Tabu list $\mathcal{L}$ as an empty set and set the current optimal number of disk seeks $q(\mathbb{F}_{opt})$ as infinity (step 1). In a new iteration, for a given current recovery solution $\mathbb{R}$ and each lost element x, we replace the current selected parity chain S_x with each candidate parity chain C_x, and construct another recovery solution $\mathbb{R}'$ (steps $2 \sim 5$). If $\mathbb{R}'$ retrieves more than M elements, then it will be optimized to be a valid one by reusing Algorithm 1 (steps $6 \sim 7$). Given $\mathbb{R}'$, we can have the distribution of retrieved elements, which is denoted as $\mathbb{D}'$ (step 8). We run the Filling Procedure (i.e., Algorithm 2) to fill the intervals in $\mathbb{D}'$, obtain the element distribution $\mathbb{F}'$, and calculate the number of seek operations $q(\mathbb{F}')$ (steps $9 \sim 10$). If $q(\mathbb{F}')$ is not kept in the Tabu list, then we record the tuple $\{\mathbb{R}', \mathbb{F}'\}$ in the candidate set $\mathcal{A}$ (steps $11 \sim 12$).

After testing every candidate parity chain for each lost element, we select the tuple $\{\mathbb{R}_{min}, \mathbb{F}_{min}\}$ from $\mathcal{A}$, where $\mathbb{F}_{min}$ has the fewest disk seeks among the solutions in $\mathcal{A}$ (step 13). We compare $q(\mathbb{F}_{min})$ with the optimal number of seek requests $q(\mathbb{F}_{opt})$ currently found, and record $\mathbb{F}_{min}$ and $q(\mathbb{F}_{min})$ if $\mathbb{F}_{min}$ brings fewer seek operations than $\mathbb{F}_{opt}$ (steps



(a) Element Distribution Before Replacement. It needs 24 elements and causes 17 seek operations.

(b) Element Distribution After Replacement. It needs 25 elements and causes 15 seek operations.

(c) Element Distribution After Filling. It needs 27 elements and causes 13 seek operations.

Fig. 6. An example for a replacement in Algorithm 3.

$15 \sim 17$). Finally, we append $q(\mathbb{F}_{min})$ in the Tabu list $\mathcal{L}$ (step 21) and select $\mathbb{R}_{min}$ for the next iteration (step 22). As Algorithm 3 proceeds, $q(\mathbb{F}_{opt})$ will be iteratively reduced.

An Example. The current recovery solution in Fig. 4c is $\mathbb{R} := (\mathcal{R}_1, \mathcal{R}_2)$, where $\mathcal{R}_1 := \{E_{1,5}, E_{2,6}, E_{3,6}, E_{4,5}\}$ and $\mathcal{R}_2 := \{E_{6,5}, E_{6,4}, E_{8,5}, E_{8,4}\}$. This solution requires 24 elements and causes 17 disk seeks. The element distribution of $\mathbb{R}$ is shown in Fig. 6a.

We set $M = 27$ and assume $\mathcal{L} = \{14\}$, which is updated in previous iterations. The current parity chain for $E_{1,1}$ (i.e., element x in Algorithm 3) in $\mathbb{R}$ is the horizontal parity chain led by $E_{1,5}$ (i.e., S_x). We replace it with another parity chain that $E_{1,1}$ involves (i.e., the diagonal parity chain led by $E_{1,6}$ (denoted by C_x)), and construct a candidate solution $\mathbb{R}' = (\mathcal{R}'_1, \mathcal{R}_2)$, where $\mathcal{R}'_1 := \{E_{1,6}, E_{2,6}, E_{3,6}, E_{4,5}\}$.

We can obtain a new element distribution $\mathbb{D}'$ as shown in Fig. 6b. $\mathbb{D}'$ is a valid distribution because it requires 25 elements ($\leq M = 27$). We then calculate the number of filling elements (i.e., $\Delta = 2$), perform the filling procedure (i.e., read unrequested data elements $E_{5,2}$ and $E_{7,3}$), and obtain the element distribution (i.e., $\mathbb{F}'$) as shown in Fig. 6c. The number of disk seeks in $\mathbb{F}'$ (i.e., $q(\mathbb{F}')$) is 13. Since $13 \notin \mathcal{L}$, we record the tuples $\{\mathbb{R}', \mathbb{F}'\}$ in $\mathcal{A}$.

After testing all the possible candidate parity chains for every lost element $\{E_{1,1}, E_{2,1}, \ldots, E_{8,1}\}$, we perform the greedy selection in steps $13 \sim 22$.

4.3 Complexity Analysis

We now analyze the complexities of Algorithms 1, 2, and 3, based on the notation listed in Tables 1 and 2.

Complexity of Algorithm 1. Since the complexity of Zpacr is $O(mw^3)$ [14] and Algorithm 1 will invoke it for at most s times, the complexity of Algorithm 1 is $O(smw^3)$.

Complexity of Algorithm 2. The total number of intervals is no more than $(n-1)sw$, where $(n-1)$ is the number of surviving disks and $(n-1)sw$ is the total number of surviving elements. Therefore, the sorting complexity is $O(nsw\log(nsw))$ when employing the sorting algorithms, such as Quicksort [36] (step 2 in Algorithm 2). For each interval to be padded, Algorithm 2 should scan every surviving disks (steps $3 \sim 9$). The complexity is no more than $O(n^2sw)$. Therefore, the complexity of Algorithm 2 is $O(nsw\log(nsw)) + O(n^2sw)$.

Complexity of Algorithm 3. In Algorithm 3, for each parity replacement, Algorithm 3 may reuse Algorithm 1 and invoke Algorithm 2. Since our simplified recovery model only considers the parity chains that generate parity elements, there are at most mw parity chains in a stripe. Each iteration will try at most smw replacements for s stripes, so the total number of replacements after t iterations is at most $tsmw$. Based on the above analysis, in Algorithm 3, the complexity of calling Algorithm 2 is $O(tnms^2w^2\log(nsw)) + O(tmn^2s^2w^2)$, while the complexity of reusing Algorithm 1 is $O(ts^2m^2w^4)$. Therefore, the complexity of Algorithm 3 is $O(ts^2m^2w^4) + O(tnms^2w^2\log(nsw)) + O(tmn^2s^2w^2)$.

5 Performance Evaluation

We conduct a series of intensive tests to evaluate the performance of SIOR. We choose the stripe-level greedy algorithm Zpacr [14] as the baseline, which also works for any XOR-based erasure code but only optimizes the number of read elements without considering the optimization of disk seeks. Therefore, the comparison can fairly represent the advantage of SIOR.

We select four typical coding schemes, including RDP Code (over $p+1$ disks, where p is a prime number), X-Code (over p disks), STAR Code (over $p+3$ disks) and CRS Code. Table 3 shows the properties of the four coding schemes, where n is the number of disks in a stripe and m is the tolerable number of disk failures.

Evaluation Environment. We choose n from 5 to 15. This range covers typical system configurations of many well-known storage systems [3], [37]. The test is run on a Linux server with a X5472 processor and 8 GB memory. The operating system is SUSE Linux Enterprise Server and the filesystem is EXT3. The deployed disk array consists of 15 Seagate/

TABLE 3
Four Representative Coding Schemes
(p is a Prime Number)

Coding Scheme	n	k	m
RDP Code	$p+1$	$p-1$	2
X-Code	p	$p-2$	2
STAR Code	$p+3$	p	3
CRS Code	general pairs of (k, m)		

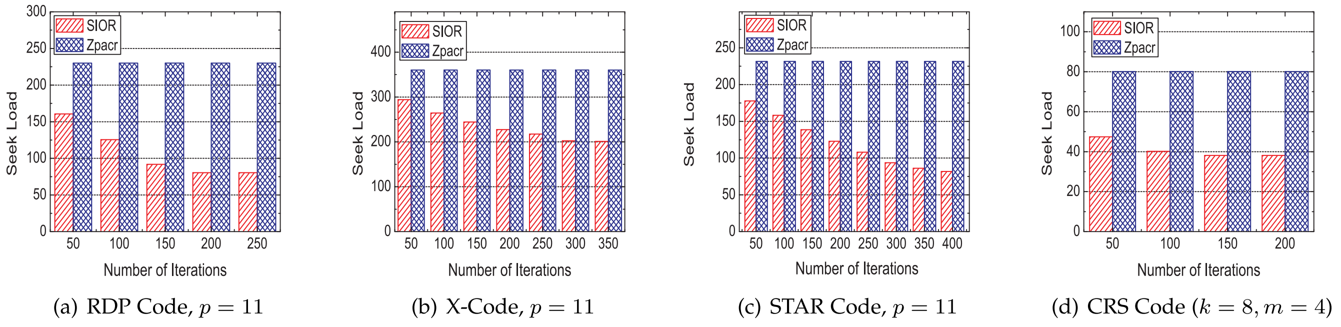


Fig. 7. The seek load under different numbers of iterations. The smaller value means the lighter load on disks.

Savvio 10K.3 SAS disks, each of which has 300 GB storage capability and 10,000 rmp. We organize the disks in the just a bunch of disks (JBOD) mode and each disk is independently handled as a node. The machine and the disk array are connected by a Fiber cable with the bandwidth of 800 MB/sec. The codes are realized by Jerasure 1.2 [32], a widely-used library to realize erasure coding storage systems.

Evaluation Metrics. Suppose the storage system consists of n disks $\{D_1, \ldots, D_n\}$. When D_i fails, a recovery solution causes o_j disk seeks to D_j $(1 \leq j \neq i \leq n)$. Suppose the recovery time (i.e., the time to read the wanted data from surviving disks, recover the lost data, and store the reconstructed data at a new disk) is T. We concern the following metrics:

1) *Seek load*. This metric denotes the average number of seek operations that a surviving disk serves during the reconstruction and is defined as follows:

$$\text{seek load} = \frac{\sum_{1 \leq j \neq i \leq n} o_j}{n-1}. \tag{3}$$

2) *Average recovery time*. This metric denotes the time that is needed to repair a certain amount of data. We mainly consider the average recovery time per MB, which is denoted as the ratio of the recovery time T and the size of repair data (in MB)

$$\text{average recovery time} = \frac{T}{\text{size of repaired data}}. \tag{4}$$

3) *Recovery speed*. As the reciprocal of recovery time, the recovery speed can then be defined as follows:

$$\text{recovery speed} = \frac{\text{size of repaired data}}{\text{recovery time}}. \tag{5}$$

Evaluation Methodology. For each coding scheme, we generate the data elements and perform the encoding by producing parity elements. These elements are then dispersed according to the layout of that code and the stripe rotation principle. The element size is set as 16 KB and the stripe number is set as 100. We then destroy the elements on a random selected disk, and trigger SIOR to generate the recovery solution over these 100 stripes. We also employ Zpacr [14] to compute the recovery solution for each stripe. Finally, we repair the lost elements based on the solutions found by SIOR and Zpacr respectively. To recover the failed disk, for each solution, the system will perform the following steps: 1) retrieving the needed data from the surviving disks; 2) performing the data reconstruction; and 3) writing the reconstructed data to a new disk.

5.1 Impact of Number of Iterations

In this experiment, we evaluate the performance on seek load and average recovery time when the number of iterations increases. We select $p = 11$, so that the number of disks n constructed over RDP Code, X-Code and STAR Code will be 12, 11 and 13, respectively. For CRS Code, we choose the parameter $(k = 8, m = 4)$, so that the number of disks $n = k + m = 12$. Like in Zpacr [14], we also select $w = 6$ for CRS Code. For each code, SIOR reads 5 percent more elements than Zpacr. The test results are shown in Figs. 7 and 8.

Seek Load. Fig. 7 demonstrates that SIOR can significantly reduce the seek load for various kinds of codes, when compared to Zpacr [14]. For RDP Code, SIOR cuts down up to 65.1 percent of seek operations for each disk during the reconstruction. For X-Code, SIOR decreases up to 44.2 percent of seek operations. For STAR Code and CRS Code, SIOR removes up to 64.7 and 52.4 percent of seek operations, respectively.

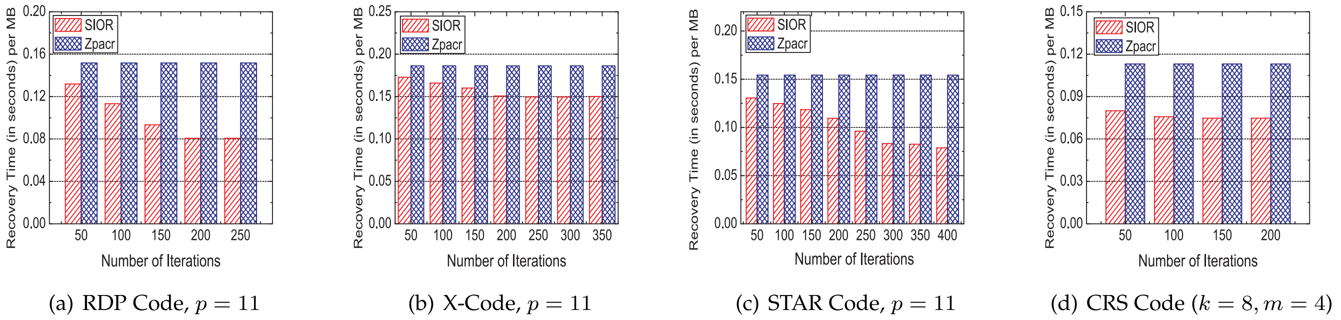
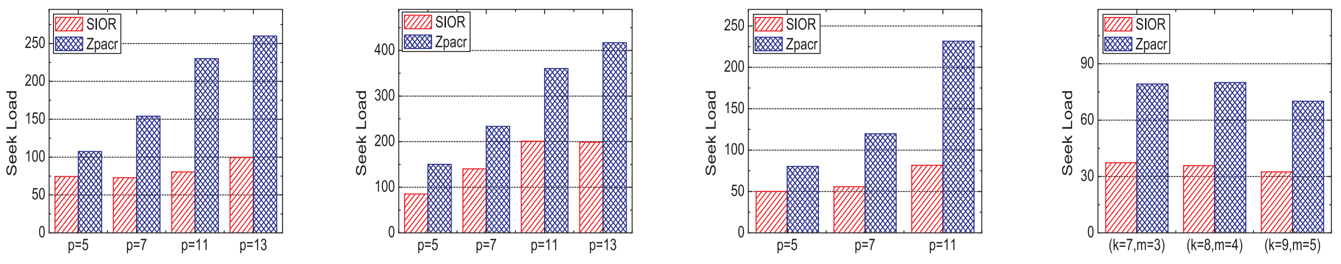


Fig. 8. The average recovery time (in seconds) per MB under different numbers of iterations. The larger value means the more needed time for data reconstruction.



(a) RDP Code (b) X-Code (c) STAR Code (d) CRS Code

Fig. 9. The seek load under different number of disks. The smaller value means the lighter load on disks.

Moreover, with the increasing of iterations, the number of optimized seek operations first sharply decreases and then becomes stable. This is because the gained reduction on seek operations in each iteration generally becomes smaller when the sought solution is closer to the optimal one.

Average Recovery Time. Fig. 8 shows that SIOR significantly decreases the average recovery time when compared with Zpacr. For RDP Code, the recovery solution found by SIOR needs 46.8 percent less time than that sought by Zpacr. For STAR Code and CRS Code, SIOR cuts down the average recovery time by up to 49.0 and 34.0 percent respectively compared to Zpacr [14]. Meanwhile, similar with the tendency of disk seeks, the average recovery time will first be rapidly decreased with the increase of iteration steps and then become stable. This phenomenon demonstrates the benefit brought by the reduction on disk seeks. Besides, the average recovery time evaluated in our experiment is similar to that in [6] and [33].

5.2 Scalability

In this experiment, we evaluate the scalability of SIOR in terms of seek load and average recovery time when the system scale expands. The number of iterations is set as 400. In the evaluation of each code, SIOR reads 5 percent more elements than Zpacr. For RDP Code, X-Code, and STAR Code, we measure the seek load and average recovery time under different selections of p. For CRS Code, we select different parameter pairs of (k, m). Results are shown in Figs. 9 and 10.

Seek Load. Fig. 9 indicates that SIOR keeps its advantage to optimize disk seeks under different scales of the storage systems. Take RDP Code as an example, SIOR decreases about 31.8 percent of seek operations loaded on each disk when $p = 5$, and this reduction increases to 62.8 percent when $p = 13$.

Moreover, SIOR widens the benefit on disk seeks reduction when the scale of the storage system expands. Take STAR Code as an example, the saving of seek operations brought by SIOR increases from 37.7 percent ($p = 5$) to 64.7 percent ($p = 11$).

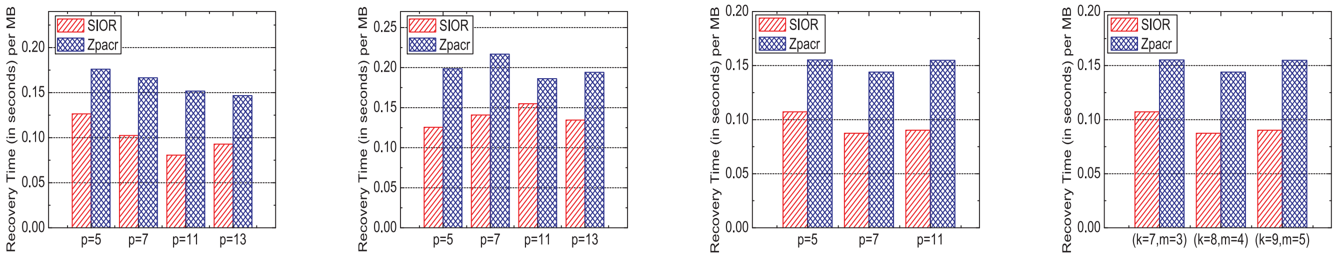
Average Recovery Time. Fig. 10 confirms that SIOR keeps its capability to shorten the average recovery time under different system scales, as SIOR still behaves well to reduce the caused seek operations when the system scale expands. For example, for RDP Code, the recovery solution found by SIOR needs 38.5 percent less time compared with that sought by Zpacr when $p = 7$, which also indicates that the recovery speed brought by SIOR is 62.5 percent faster than that by using Zpacr when $p = 7$.

We have another observation that the reduction on average recovery time brought by SIOR will be different when the system scale varies. For example, for X-Code, the recovery solution by using SIOR requires 16.9 percent less time for data reconstruction when compared with that found by Zpacr. The time saving will expand to 36.8 percent when $p = 5$.

5.3 Impact of Expected Retrieved Data

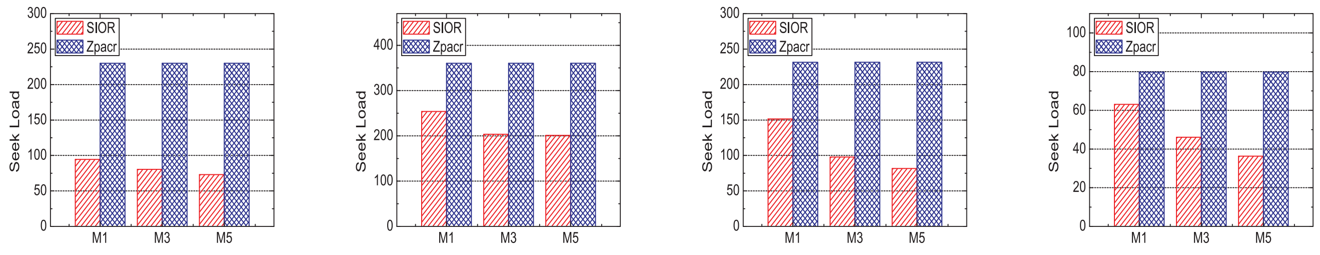
We further evaluate the seek load and average recovery time of SIOR under different number of elements that are expected to be read. The number of iterations is set as 400. We select $p = 11$, so that the number of disks n constructed over RDP Code, X-Code and STAR Code will be 12, 11 and 13, respectively. For CRS Code, we choose the parameter $(k = 8, m = 4)$, so that the number of disks $n = k + m = 12$. To find the recovery solution of each code, SIOR consider three different selections M, i.e., $M_1 = (1 + 1\%) \times C_{min}$, $M_3 = (1 + 3\%) \times C_{min}$, and $M_5 = (1 + 5\%) \times C_{min}$, where C_{min} is the least number of required elements found by Zpacr [14]. The results are shown in Figs. 11 and 12.

Seek Load. Fig. 11 indicates that the seek load on each disk will be lighter when the number of elements that are expected to be read increases. For STAR Code, when $M = M_1$, the recovery solution found by SIOR requires



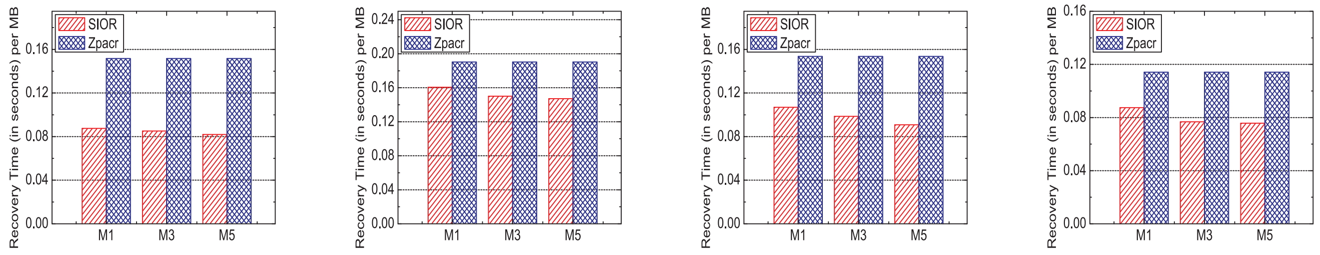
(a) RDP Code (b) X-Code (c) STAR Code (d) CRS Code

Fig. 10. The average recovery time (in seconds) per MB under different number of disks. The larger value means the more needed time for data reconstruction.



(a) RDP Code, $p = 11$ (b) X-Code, $p = 11$ (c) STAR Code, $p = 11$ (d) CRS Code ($k = 8, m = 4$)

Fig. 11. The seek load under different expected number of read elements. The smaller value means the lighter load on disks.



(a) RDP Code, $p = 11$ (b) X-Code, $p = 11$ (c) STAR Code, $p = 11$ (d) CRS Code ($k = 8, m = 4$)

Fig. 12. The average recovery time (in seconds) per MB under different expected number of read elements. The larger value means the more needed time for data reconstruction.

about 34.5 percent less disk seeks compared with that found by Zpacr. The reduction will increase to 57.7 percent when $M = M_3$. This reduction should owe to two reasons. First, when given a larger number of elements that are expected to be read, SIOR has a larger search space to find the recovery solution with less disk seeks. Second, for a same recovery solution, when the number of elements expected to be read is larger, there will be more elements that are allowed to be filled for interval padding, which then facilitates the reduction of disk seeks.

Average Recovery Time. Fig. 12 shows that SIOR can shorten the average recovery time when the number of expected elements increases. For RDP Code, when $M = M_1$, the recovery time (per MB) using SIOR will be 42.3 percent less than that employing Zpacr. When $M = M_3$, the time saving will increase to 43.9 percent. We can observe that the average recovery time will not be greatly reduced when $M = M_5$. More expected elements though facilitate the search of the recovery solution with less disk seeks, it also causes more repair traffic (i.e., the amount of data to be read for data recovery) in reverse. Therefore, the selection of M should be carefully considered in practice.

5.4 Accuracy and Efficiency

We then evaluate the accuracy and efficiency of SIOR. Accuracy is used to denote the closeness between the optimal solutions sought by SIOR and the Enumeration respectively

$$\text{accuracy} = \frac{\text{the least disk seeks in SIOR}}{\text{the least disk seeks in Enumeration}}. \quad (6)$$

We select RDP Code ($p = 11$), X-Code ($p = 11$), and STAR Code ($p = 11$), and vary the number of stripes. In the evaluation of each code, both SIOR and Enumeration method read 5 percent more elements than Zpacr. The number of iteration steps is set as 400. We run SIOR and Enumeration method respectively, calculate the search accuracy, and record the search latency in Table 4.

Table 4 indicates that SIOR is accurate and efficient when compared with Enumeration. SIOR obtains the solution with the same minimum number of disk seeks compared to Enumeration in RDP Code, and causes no more than 8 percent extra seek operations when compared with Enumeration in X-Code. On the aspect of efficiency, SIOR greatly decreases the search latency. For example, SIOR only needs 0.21 seconds to find the solution with the least disk seeks for RDP code when the number of stripes is 3. On the contrary, the search time of Enumeration exponentially enlarges when the number of stripes increases. For example, for RDP Code, Enumeration requires more than 3 hours to find the solution with minimum seek operations when there are only three stripes.

5.5 Load Balancing

To evaluate the effect on load balancing, we also compare the new proposed SIOR in this paper with that in the preliminary version [1]. Like in [10], [11], to reflect the

TABLE 4
The Accuracy and Efficiency of SIOR

Num. of Stripe	Accuracy	Time (SIOR)	Time (Enumeration)
RDP Code ($p = 11$)			
1	1.00	0.04 sec	0.04 sec
2	1.00	0.12 sec	7.52 sec
3	1.00	0.21 sec	3h 17min 30sec
X-Code ($p = 11$)			
1	1.05	0.03 sec	0.04 sec
2	1.08	0.15 sec	26.04 sec
3	1.02	0.27 sec	20h 4min 8sec
STAR Code ($p = 11$)			
1	1.00	0.21 sec	0.22 sec
2	1.09	0.29 sec	7h 1min 10sec

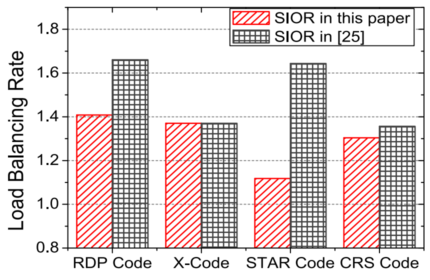


Fig. 13. Comparison on load balancing rate.

balancing degree of the load, we first define the concept of *load balancing rate* by using the ratio of the number of read elements at the most loaded disk with that at the least loaded disk. In this test, we select $p = 7$ for RDP Code, X-Code, and STAR Code, respectively. We also choose the parameters $(k = 7, m = 3, w = 6)$ for CRS Code. For each code, SIOR reads 5 percent more elements than Zpacr. The test results are shown in Fig. 13.

We can observe that compared with the preliminary version [1], the proposed SIOR in this paper behaves better on evenly dispersing the element requests across the surviving disks during the recovery. For example, for STAR Code, the load balancing rate in the preliminary version [1] will be 1.64. As a comparison, after applying SIOR in this paper, the load balancing rate will be decreased to 1.12. This is because the filling procedure of SIOR in this paper will selectively increase the load at the lightest disk, and thus narrow the gap between the most loaded disk and the least loaded one.

5.6 Data Reduction

We also test the capability of SIOR to reduce the amount of read data for recovery. We select $p = 11$ for RDP Code, X-Code, and STAR Code. We also use CRS Code with the parameters $(k = 8, m = 4, w = 6)$. In the evaluation of each code, SIOR reads 5 percent more elements than Zpacr. The comparison results among Zpacr, SIOR, and the conventional method (i.e., without hybrid parity chains) are normalized in Table 5. Table 5 indicates that both Zpacr and SIOR impressively decrease the amount of read data compared to the conventional method. For example, SIOR reduces 19 percent unnecessary retrieved elements compared with the conventional method. Meanwhile, SIOR only retrieves 5 percent more elements compared with Zpacr, and this ratio can also be further adjusted according to the administrator's requirements.

TABLE 5
Comparison on Data Reduction

Codes	Zpacr	SIOR	Conventional Method
RDP Code	0.77	0.81	1.00
X-Code	0.72	0.76	1.00
STAR Code	0.72	0.76	1.00
CRS Code	0.87	0.91	1.00

5.7 Impact of Element Size

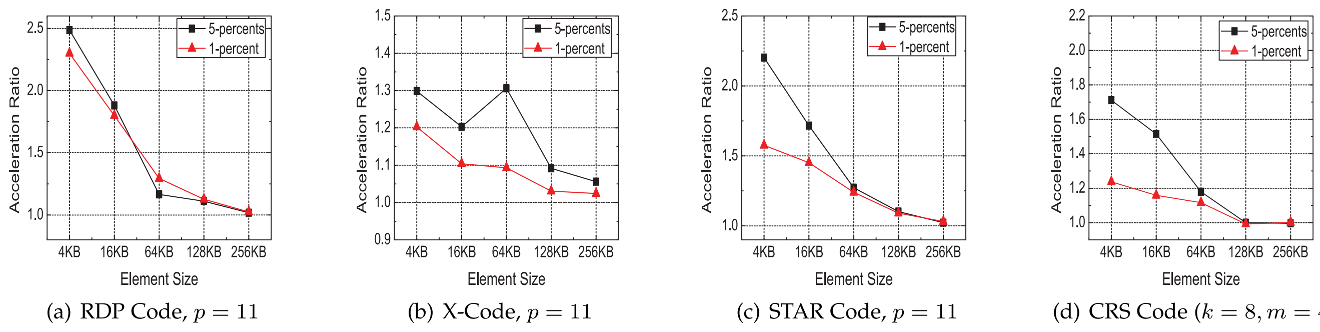
To investigate how the element size affects the benefit of SIOR, we compare SIOR with Zpacr in terms of recovery speed under different element sizes. We first define the concept of *acceleration ratio* as

$$\text{acceleration ratio} = \frac{\text{recovery speed of SIOR}}{\text{recovery speed of Zpacr}}. \tag{7}$$

Obviously, when the acceleration ratio is larger, SIOR can achieve a faster data reconstruction when compared with Zpacr [14]. With respect to the selection of M, we mainly consider two cases for each code, i.e., $M_1 = (1 + 1\%) \times C_{min}$ and $M_5 = (1 + 5\%) \times C_{min}$, where C_{min} is the least number of required elements found by Zpacr [14]. We also select RDP Code $(p = 11)$, X-Code $(p = 11)$, STAR Code $(p = 11)$, and CRS Code $(k = 8, m = 4)$ in this test. The number of iteration steps is set as 400. We then vary the element size from 4 to 256 KB, and calculate the acceleration ratio under these two selections of M (i.e., $M = M_1$ and $M = M_5$). The results are presented in Fig. 14.

First, the advantage of SIOR will eliminate when the element size increases. For example, when the element size is 4 KB and $M = (1 + 5\%) \times C_{min}$, the acceleration ratio of RDP Code is 2.5, meaning that the recovery speed of SIOR is 1.5 times faster than that of Zpacr. When the element size increases to 256 KB, SIOR reaches almost the same recovery speed with Zpacr. This test also suggests that SIOR is more suitable to be deployed in the environment with a small element size (i.e., smaller than 256 KB). This is because the influence of seek time declines when the size of I/O unit expands.

Second, the number of read elements also affects the performance of SIOR when the element size is larger. When the element size is small, selecting $M = M_5$ gains more performance improvement, compared with the chosen of $M = M_1$. However, when the element size increases, the advantage of selecting $M = M_5$ will drop. This is because the effect of disk seek reduction becomes insignificant and the increasing size of retrieved elements also slows down the reconstruction in reverse.



(a) RDP Code, $p = 11$ (b) X-Code, $p = 11$ (c) STAR Code, $p = 11$ (d) CRS Code $(k = 8, m = 4)$

Fig. 14. The acceleration rate under different element sizes.

5.8 Summary

These tests show that both Zpacr and SIOR are suitable for storage systems that are easily restricted by the repair traffic, such as [12] and [38]. Compared with Zpacr, SIOR not only sustains the advantage of data reduction, but also decreases disk seeks to reach the faster data reconstruction.

6 Conclusion

Previous studies on single failure recovery confine to the minimization of retrieved data, and overlook the influence of disk seeks. In this paper, we propose a greedy algorithm called SIOR based on Tabu search to optimize both the number of disk seeks and the amount of read data for recovery. The algorithm includes two stages. The first stage makes use of an existing algorithm that optimizes the number of retrieved elements for each stripe, and gets an initial solution. The second stage optimizes the initial solution and iteratively approaches the optimal recovery solution. Finally, the evaluation indicates that SIOR reduces $31.8 \sim 65.1$ percent of disk seeks and improves the recovery speed by up to 150.0 percent during the recovery.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (Grant No. 61232003, 61327902, 61433008), University Grants Committee of Hong Kong (Grant No. AoE/E-02/08), VC Discretionary Fund of CUHK (Grant No. VCF2014007), and Research Committee of CUHK. Jiwu Shu (shujw@tsinghua.edu.cn) is the corresponding author. A preliminary version[1] of this paper was presented at the 34th International Symposium on Reliable Distributed Systems (SRDS'15). In this journal version, we include additional analysis on load balancing and more evaluation results for SIOR.

References

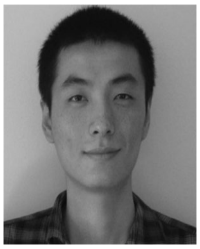
[1] Z. Shen, J. Shu, and Y. Fu, "Seek-efficient I/O optimization in single failure recovery for XOR-coded storage systems," in *Proc. IEEE 34th Symp. Reliable Distrib. Syst.*, 2015, pp. 228–237.
[2] B. Schroeder and G. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?" in *Proc. 5th USENIX Conf. File Storage Technol.*, 2007, Art. no. 1.
[3] C. Huang, et al., "Erasure coding in windows azure storage," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, pp. 15–26.
[4] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 331–342.
[5] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster," in *Proc. 5th USENIX Conf. Hot Topics Storage File Syst.*, 2013, pp. 8–8.
[6] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads," in *Proc. USENIX Fast Storage Technol.*, pp. 251–264, 2012.
[7] L. Xiang, Y. Xu, J. Lui, and Q. Chang, "Optimal recovery of single disk failure in RDP code storage systems," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Syst.*, 2010, pp. 119–130.
[8] L. Xu and J. Bruck, "X-code: MDS array codes with optimal encoding," *IEEE Trans. Inf. Theory*, vol. 45, no. 1, pp. 272–276, Jan. 1999.
[9] X. Luo and J. Shu, "Load-balanced recovery schemes for single-disk failure in storage systems with any erasure code," in *Proc. IEEE 42nd Int. Conf. Parallel Process.*, 2013, pp. 552–561.
[10] C. Wu, et al., "HDP Code: A horizontal-diagonal parity code to optimize I/O load balancing in RAID-6," in *Proc. IEEE/IFIP 41st Int. Conf. Depend. Syst. Netw.*, 2011, pp. 209–220.
[11] Z. Shen and J. Shu, "HV Code: An all-around MDS code to improve efficiency and reliability of RAID-6 systems," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2014, pp. 550–561.
[12] Y. Hu, H. C. Chen, P. P. Lee, and Y. Tang, "NCCloud: Applying network coding for the storage repair in a cloud-of-clouds," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 21–21.
[13] S. Xu, et al., "Single disk failure recovery for X-code-based parallel storage systems," *IEEE Trans. Comput.*, vol. 63, no. 4, pp. 995–1007, Apr. 2013.
[14] Y. Zhu, P. P. Lee, Y. Hu, L. Xiang, and Y. Xu, "On the speedup of single-disk failure recovery in XOR-coded storage systems: Theory and practice," in *Proc. IEEE 28th Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–12.
[15] C. Ruemmler and J. Wilkes, "An introduction to disk drive modeling," *Computer*, vol. 27, no. 3, pp. 17–28, 1994.
[16] Y. Fu, J. Shu, and X. Luo, "A stack-based single disk failure recovery scheme for erasure coded storage systems," in *Proc. IEEE 33rd Int. Symp. Reliable Distrib. Syst.*, 2014, pp. 136–145.
[17] Z. Shen, J. Shu, and P. P. Lee, "Reconsidering single failure recovery in clustered file systems," in *Proc. 46th IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2016.
[18] (2016). EXT3. [Online]. Available: http://www.en.wikipedia.org/wiki/ext3:
[19] Z. Zhang and K. Ghose, "yFS: A journaling file system design for handling large data sets with reduced seeking," in *Proc. 2nd USENIX Conf. File Storage Technol.*, 2003, pp. 59–72.
[20] (2011). EMC CLARiiON best practices for performance and availability: Release 30.0 firmware update applied best practice. [Online]. Available: https://www.emc.com/collateral/hardware/white-papers/h5773-clariion-best-pra ctices-performance-availability-wp.pdf
[21] J. Gray, B. Horst, and M. Walker, "Parity striping of disk arrays: Low-cost reliable storage with acceptable throughput," in *Proc. 16th Int. Conf. Very Large Data Bases*, 1990, pp. 148–161.
[22] F. Glover and M. Laguna, *Tabu Search*. Berlin, Germany: Springer, 1999.
[23] J. J. Wylie and R. Swaminathan, "Determining fault tolerance of XOR-based erasure codes efficiently," in *Proc. 37th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2007, pp. 206–215.
[24] C. Jin, H. Jiang, D. Feng, and L. Tian, "P-Code: A new RAID-6 code with optimal properties," in *Proc. 23rd Int. Conf. Supercomputing*, 2009, pp. 360–369.
[25] P. Corbett, et al., "Row-diagonal parity for double disk failure correction," in *Proc. 3rd USENIX Conf. File Storage Technol.*, 2004, pp. 1–1.
[26] C. Wu, S. Wan, X. He, Q. Cao, and C. Xie, "H-Code: A hybrid MDS array code to optimize partial stripe writes in RAID-6," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 782–793.
[27] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. SIAM-8, no. 2, pp. 300–304, 1960.
[28] M. Li and P. P. Lee, "Stair codes: A general family of erasure codes for tolerating device and sector failures in practical storage systems," in *Proc. 12th USENIX Conf. File Storage Technol.*, 2014, pp. 147–162.
[29] (2009). EMC symmetrix DMX-RAID 6 implementation. [Online]. Available: http://storagenerve.com/2009/02/27/emc-symmetrix-dmx-raid-6-implementation/
[30] C. Lueth, "RAID-DP: Network appliance implementation of raid double parity for data protection," Netw. Appliance, Inc., Sunnyvale, CA, USA, *Tech. Rep. TR-3298*, 2004.
[31] G. Zhang, G. Wu, J. Shu, K. Li, S Wang, and W. Zheng, "CaCo: An efficient cauchy coding approach for cloud storage systems," *IEEE Trans. Comput.*, vol. 65, no. 2, pp. 435–447, Feb. 2015.
[32] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2," Dep. Elec. Eng. Comput. Sci., Univ. Tennessee, Knoxville, TN, USA, Tech. Rep. CS-08–627, 2008, vol. 23.
[33] Y. Zhu, P. P. Lee, L. Xiang, Y. Xu, and L. Gao, "A cost-based heterogeneous recovery scheme for distributed storage systems with RAID-6 codes," in *Proc. 42nd Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2012, pp. 1–12.
[34] Z. Wang, A. Dimakis, and J. Bruck, "Rebuilding for array codes in distributed storage systems," in *Proc. IEEE Globecom Workshops*, 2010, pp. 1905–1909.

[35] M. Holland, G. Gibson, and D. Siewiorek, "Architectures and algorithms for on-line failure recovery in redundant disk arrays," *Distrib. Parallel Databases*, vol. 2, no. 3, pp. 295–335, 1994.

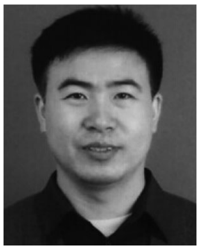
[36] C. A. Hoare, "Quicksort," *Comput. J.*, vol. C-5, no. 1, pp. 10–16, 1962.

[37] (2010). HDFS RAID. [Online]. Available: https://wiki.apache.org/hadoop/HDFS-RAID

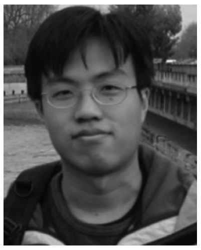
[38] Z. Wilcox-O'Hearn and B. Warner, "Tahoe: The least-authority filesystem," in *Proc. 4th ACM Int. Workshop Storage Secur. Survivability*, 2008, pp. 21–26.



Zhirong Shen received the bachelor's degree from the University of Electronic Science and Technology of China, and the PhD degree from the Department of Computer Science and Technology, Tsinghua University in 2016. He is now an assistant professor at Fuzhou University. His current research interests include storage reliability and storage security.



Jiwu Shu received the PhD degree in computer science from Nanjing University in 1998, and finished the postdoctoral position research at Tsinghua University in 2000. Since then, he has been teaching at Tsinghua University. His current research interests include storage security and reliability, non-volatile memory based storage systems, and parallel and distributed computing. He is a member of the IEEE.



Patrick P.C. Lee received the BEng degree (first-class honors) in information engineering from Chinese University of Hong Kong in 2001, the MPhil degree in computer science and engineering from Chinese University of Hong Kong in 2003, and the PhD degree in computer science from Columbia University in 2008. He is now an associate professor in the Department of Computer Science and Engineering, Chinese University of Hong Kong. His research interests include in cloud storage, distributed systems and networks, and security/resilience.

Yingxun Fu received the bachelor's degree from North China Electric Power University in 2007, the master's degree from Beijing University of Posts and Telecommunications in 2010, and the doctor's degree from Tsinghua University in 2015. He is now an assistant professor at the North China University of Technology. His current research interests include storage reliability and distributed systems.